This is the 2nd post in this series. We are exploring building a message queue using Racket. The message queue is a program that provides an API to store and retrieve messages. It organizes messages on topics and each topic has a queue associated with it.

We are using Racket’s web stack to provide the web interface, a Racket hash-table to provide the topics and a Racket data/queue to provide the queues for storing messages. Racket provides a nice JSON library for handling JSON.

In the first installment of this [series (see the file “web3-p.rkt”)]( https://github.com/muguira-james/racket-stuff.git), we described the scaffolding for the application. The application at this point exposes an API with 1 method in it: “hello”, which just provides a string with the date and time embedded when you call it. We will define more data structures and 2 more API calls in this article.

So far, our backend architecture defines a front door and a middle layer. The front door code hides the web mechanics of handling http requests. It dispatches to our simple “hello” API method. Let’s extend that to expose the dispatching. Once we have the dispatch code exposed, we can further extend to add the various API calls that finish off the application. The last line of the application called the web server start up code and established the ports, URLs and other items needed to make our sever. Our extensions follow:

```

#lang racket

(require web-server/servlet)

(require web-server/servlet-env)

(require json)

(require data/queue)

(require gregor)

(define (hello request)

(http-response "Hello from message queue"))

(define (http-response content)

(response/full

200 ; HTTP response code.

#"OK" ; HTTP response message.

(current-seconds) ; Timestamp.

TEXT/HTML-MIME-TYPE ; MIME type for content.

'() ; Additional HTTP headers.

(list ; Content (in bytes) to send to the browser.

(string->bytes/utf-8 content))))

(define-values (dispatch generate-url)

;; URL routing table (URL dispatcher).

(dispatch-rules

[("") do-nothing]

[("hello") greeting-page] ; check to see if the service is working

[("enque") #:method "post" enque]

; [("deque") #:method "post" deque]

[("topic-list") topic-list]

; [("topic-count") topic-count]

; [("topic-data") #:method "post" topic-data]

; [("drain-topic") #:method "post" drain-topic]

; [("drain-queue") #:method "post" drain-queue]

[else (error "page not found")]))

(define (request-handler request)

(dispatch request))

;; Start the server.

(serve/servlet

request-handler

#:launch-browser? #f

#:quit? #f

; have to listen on the right host NOT 127.0.0.1

#:listen-ip "0.0.0.0"

#:port 8000

#:servlet-regexp #rx"")

```

There are now 4 methods (in reverse order): server/servlet, request-handler, define-values, and http-response. Http-response has not changed. It is our common code we defined to reply to a request from a caller. It takes a string as input and create a http response suitable for a caller to process with either text (TEXT/HTML-MIME-TYPE) or JSON. To REALLY do JSON we should change the “TEXT/HTML-MIME-TYPE” string to “APPLICATION/JSON”, but we’ll leave that for now.

The next function, define-values is a Racket construct that binds variables as the language parser is working its way through code file. It builds a look up table. It works like a let statement in that variable definitions are created as the reader is parsing your Racket expressions. Racket uses a 2-step process to translate Racket expressions into working code: a reader and an expansion processor. The define-values creates and binds values to the variables during the reader process. In our case, for each item found on the input URL, the dispatcher will look for a definition. The only valid URL expansions are:

“”, which corresponds to <http://localhost:8000/>,

“hello”, which would call the hello function,

“enque”, which would call the enqueue function,

“deque”, which would call the dequeue function,

“topic-list”, which calls topic-list function,

“topic-count”, which calls the topic-count function,

“topic-data”, which calls the topic-data function,

“drain-topic”, calling the topic drain function,

“drain-que”, calling the drain queue function.

If the item decoded from the URL does equal one of those handlers, the dispatcher will call the error handler. The next function is the actual request-handler. The Racket web application framework we are working with will parse the in-coming URL and break it down into components. By the time the server is ready to call request-handler, which you notice is the 1st parameter to the server/servlet, the URL is parsed, and the API is ready to decode and route to the correct call. For example, if we were to use curl, a well-formed URL would look like:

```

$ curl --data ‘{ “param1”: “value1”, “param2”: “value2’ http://hostname/resource

```

# Middle Layer

The previous section just described the entire front door of our message queue. The next sections describe the middle layer. Here we will introduce the logic for each API call and describe how to test the code.

The first function we introduce is enqueue. The basic message storage mechanism for the program is a topic hash, which is a hash table where the keys are the topic names and each name has a value element hat is a data/queue. This function adds a message payload element into the queue associated with the topic. This topic structure can be visualized like so:

[image]

The enqueue function in our message queue system has 2 elements: the dispatch target in the front door and a function for handling the message data structure. Let’s take a look at the front door element:

```

;; a hash is structured as a topic and a queue

(define topic-hash (make-hash))

(define (enque request)

; put something in a queue

; input: { topic: "name", payload: "data-type" }

(let\* ([hsh (request->jshash request)]

[topic-name (hash-ref hsh 'topicname)]

[payload-data (hash-ref hsh 'payload)])

(begin

(add-data-to-topic topic-name payload-data)

(displayln

(format

"enq: name: ~v: data: ~v hash-size: ~v hash-keys: ~v~%"

topic-name payload-data (hash-count topic-hash) (hash-keys topic-hash)))

(let ([rtn (make-hash)])

(hash-set! rtn 'topic-name topic-name)

(hash-set! rtn 'data payload-data)

(hash-set! rtn 'count (hash-count topic-hash))

(hash-set! rtn 'keys (hash-keys topic-hash))

(displayln (with-output-to-string (lambda () (write-json rtn))))

(http-response (with-output-to-string (lambda () (write-json rtn))))))))

```

While it looks complex, that is because it handles getting the topic name and payload from the input http message. It then calls the code that handles the internal data structures. Finally, it builds the output response to send back to the client. The response is most of the code. The response creates a hash, which is converted to JSON and written to the client. The response hash fields: ‘topic-name’, ‘data’, ‘count’ and the ‘key’ value all encoded.

The enqueue function takes a topic name and a message payload element as input. Its operation is simple: if the topic-name is present, add the message data to the queue and return. If not present, create a new queue, add the message data to that queue and add that queue to the input topic-name, then return.

```

(define (add-data-to-topic key data)

;; check to see if key is in the topic-hash and add data to the correct topic

(if (contains-topic key)

(enqueue! (hash-ref topic-hash key) data)

(begin

(let ([q (make-queue)])

(enqueue! q data)

(hash-set! topic-hash key q)))))

```

Let’s test the program. To do so, use the racket program interpreter to run the front-door.rkt file. This will produce some messages. Now, in another terminal, let’s run the test code “enq.js” which will try and add a topic and payload to the message queue system. The output looks like:

In the racket terminal:

```

$ racket front-door.rkt

Your Web application is running at http://localhost:8000.

Stop this program at any time to terminate the Web Server.

enq: name: “a-topic”: data: ‘(“brownies and ice cream”) hash-size: 1 hash-keys: ‘(“a-topic”)

{“count”: 1, “data”: [“brownies and ice cream”], “keys”: [“a-topic”], “topic-name”: “a-topic” }

```

In the node code terminal:

```

$ node enq.js -q a-topic -p “brownies and ice cream”

Options-> { que\_name: ‘a-topic’, payload: ‘brownies and ice cream’ }

Sending… { que\_name: ‘a-topic’, payload: ‘brownies and ice cream’ }

{

count: 1,

data: ‘brownies and ice cream’,

keys: [ ‘a-topic’ ],

‘topic-name’: ‘a-topic’

}

```

Quite a lot of output from each window, but you can see how the racket front-door program used “display” to send data to the console.

This is the node program enq.js:

```

const fetch = require('node-fetch')

const commandlineargs = require('command-line-args')

const optionDefinitions = [

{ name: 'que\_name', alias: 'q', type: String },

{ name: 'payload', alias: 'p', type: String }

]

const options = commandlineargs(optionDefinitions)

console.log("options->", options)

var deft\_q = "james";

var deft\_payload = [ "cooking", "slacking", "hacking" ];

if (Object.keys(options).length != 0) {

deft\_q = options.que\_name

deft\_payload = options.payload

}

var data = {

topicname: deft\_q,

payload: deft\_payload

}

console.log("sending...", data)

fetch('http://localhost:8000/enque', {

method: 'post',

body: JSON.stringify(data),

headers: { 'Content-Type' : 'application/json' },

})

.then(res => res.json())

.then(json => console.log(json));

```

# Conclusion

This article described the front-door code for the message queue and demonstrated what the enqueue function would look like. The fundamental data structure of the message queue is a hash table called topic-hash. The system is composed of 3 parts: the front-door dispatcher, the data structures to store messages on topics and the middle ware code to manipulate that data structure. The code also demonstrated a node js based program to enqueue messages.